**SEMESTER 5**

**MINOR: ROBOTICS AND AUTOMATION**

**SUBJECT: WHEELED AND LEGGED ROBOTS**

**ASSIGNMENT**

**PREPARED BY**

**(22BEE051) JAIVIN PATEL – M2 BATCH**

**Problem statement:**

Develop a computer program for mobile robot simulation to show the obstacle avoidance, localization

and path planning. The following are the general guideline for each of the tasks.

**Obstacle avoidance:**

A map will be input to the code having occupied and free space (e.g. binary occupancy map). The

initial pose of the robot can be defined. Robot should move in a map without hitting with the obstacles.

Use multiple sensors in order to identify the obstacles all-round the robot.

**Localization:**

Implement the estimation using a suitable localization method. Use suitable distribution model for

sensor. The output is 2 graphical displays

a.) the map with the path of the robot overlaid

b.) the map with the (current) estimated location (x,y).

**Path planning:**

In case of path planning task, Apply the most suitable search algorithm from the starting location of

the robot to the goal location. Show the resulting tree that is searched as the algorithm proceeds.

Clearly mark the final path generated. Show the robot motion from start to the goal with the trajectory

followed by the robot.

**Code used for the given Problem statement:**

clc;

clear;

openExample('robotics/PathPlanningExample');

load exampleMaps.mat;

% Define robot parameters

robot = differentialDriveKinematics('WheelRadius', 0.05, 'WheelSpeedRange', [-inf, inf], 'TrackWidth', 0.5, 'VehicleInputs', 'VehicleSpeedHeadingRate');

initialPose = [3; 0; pi/2];

tend = 30; % End time

dt = 0.1; % Time step

t = 0:dt:tend;

pose(:, 1) = initialPose;%Define map and obstacles

map = binaryOccupancyMap(simpleMap, 2); % 10x10 map with 2 increments

setOccupancy(map, [2 2; 3 4; 5 2; 8 7; 2 9], 1); % Set obstacle coordinates

% Define sensor parameters

sensor = rangeSensor('Range', [0.1 10], 'HorizontalAngle', [-pi/3 pi/3]);

safeDistance = 0.5;

v = 0.5;

omega = 0;

% Visualization setup

figure;

TrackWidth = 0.5;

framesize = 0.8 \* TrackWidth;

vizRate = rateControl(1/dt);

%Defining Parameters for Monte Carlo Algorithm

odometryModel=odometryMotionModel

odometryModel.Noise=[0.2 0.2 0.2 0.2];

rangeFinderModel=likelihoodFieldSensorModel

rangeFinderModel.SensorLimits=[0.45 8];

rangeFinderModel.Map=map;

numParticles=1000;

initialCovariance=eye(3);

mcl=monteCarloLocalization('InitialPose',initialPose,'InitialCovariance',initialCovariance,'ParticleLimits',[500 numParticles]);

mcl.MotionModel=odometryModel;

mcl.SensorModel=rangeFinderModel;

mcl.GlobalLocalization=false;

mcl.UseLidarScan=true;

for i = 2:length(t) % Starting from 2 because we want to use initial pose

% Get sensor readings

[ranges, angles] = sensor(pose(:, i-1)' + [0.4 0.4 pi/2], map);

% Obstacle avoidance

if min(ranges) < safeDistance

omega = -0.5; % If obstacle is within safe distance, turn the robot

else

omega = 0; % Reset omega if no obstacle is near

end

% Update robot pose

vel = derivative(robot, pose(:, i-1)', [v omega]); % vel is derivative of xdot, ydot, thetadot

pose(:, i) = pose(:, i-1) + vel \* dt;

% Plot robot position and map

show(map);

hold off;

T1 = [pose(1, i); pose(2, i); 0];

plotRot = axang2quat([0 0 1 pose(3, i)]);

plotTransforms(T1', plotRot, 'MeshFilePath', 'groundvehicle.stl', 'Parent', gca, 'View', '2D', 'FrameSize', framesize);

light;

xlim([0 13]);

ylim([0 13]);

waitfor(vizRate);

end

% Plot the robot's path

figure;

show(map);

hold on;

plot(pose(1, :), pose(2, :), 'r');

hold off;

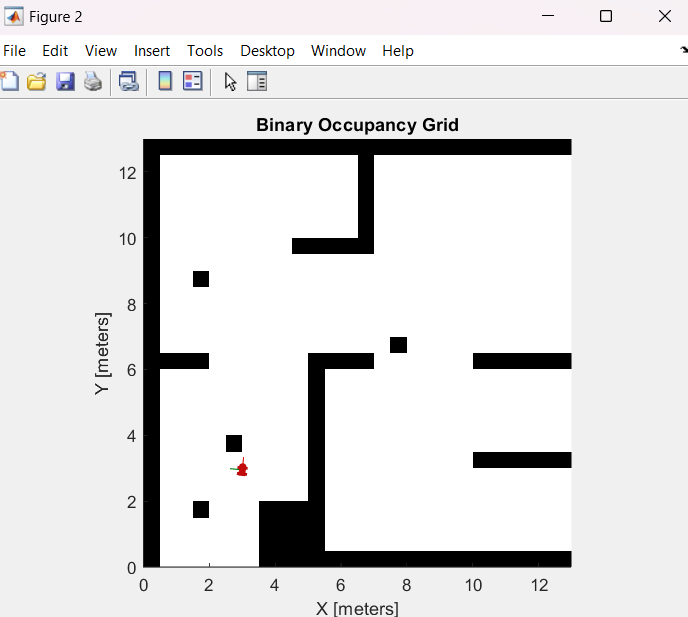
xlabel('X [m]');

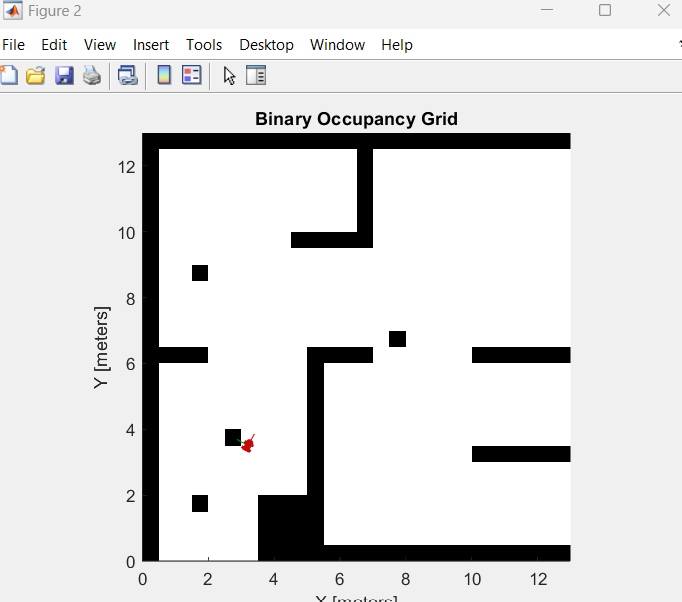
ylabel('Y [m]');

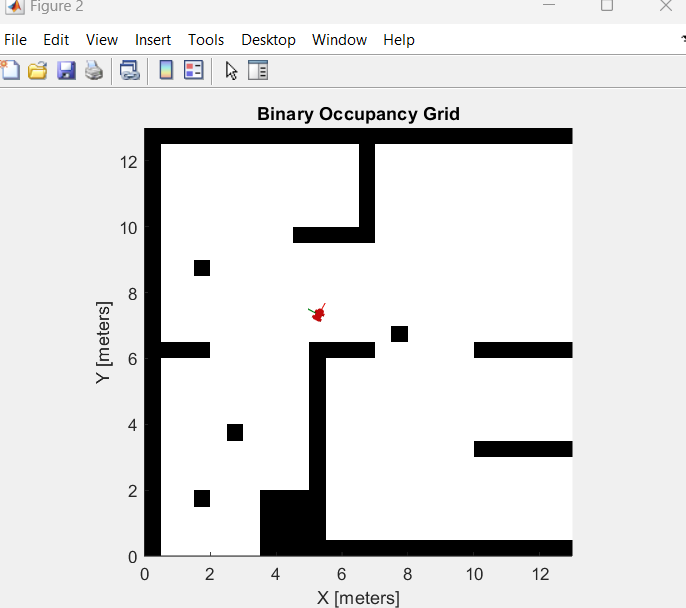
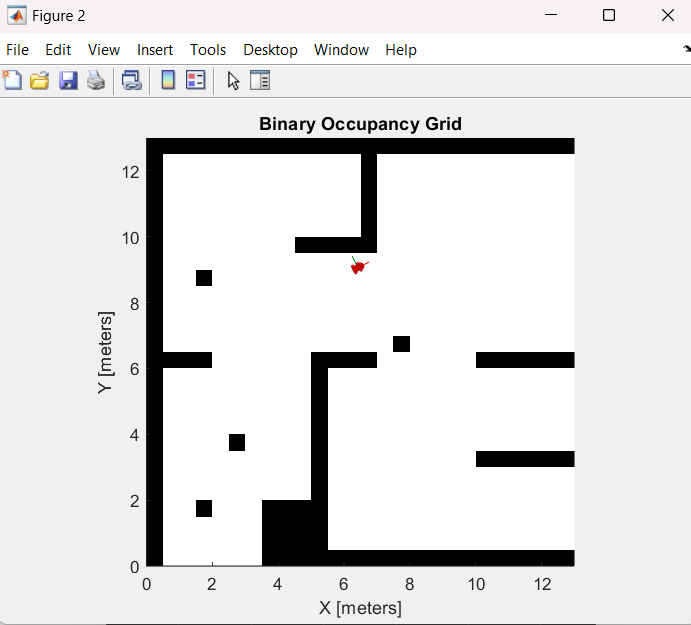
title('Robot Path with Obstacle Avoidance');

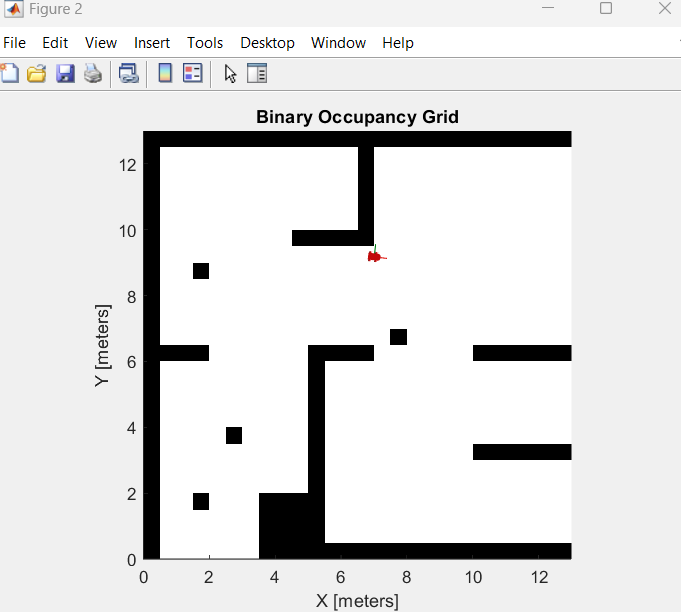
grid on;

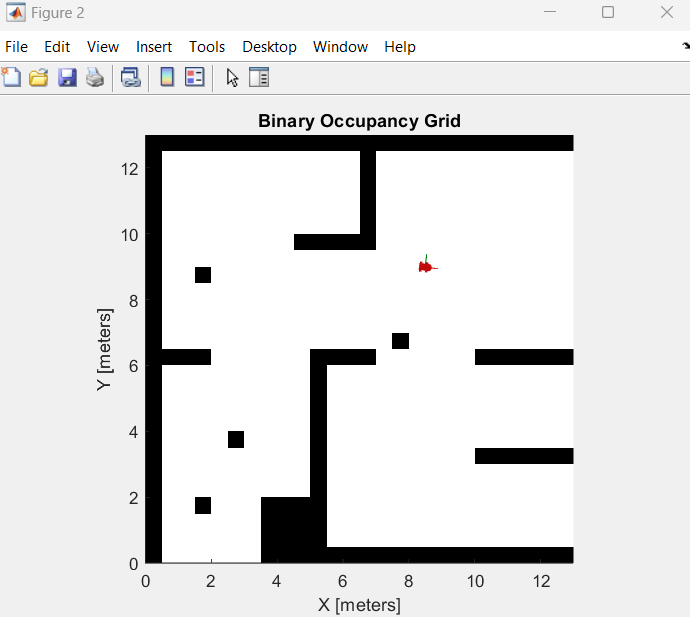
**OUTPUT:-**

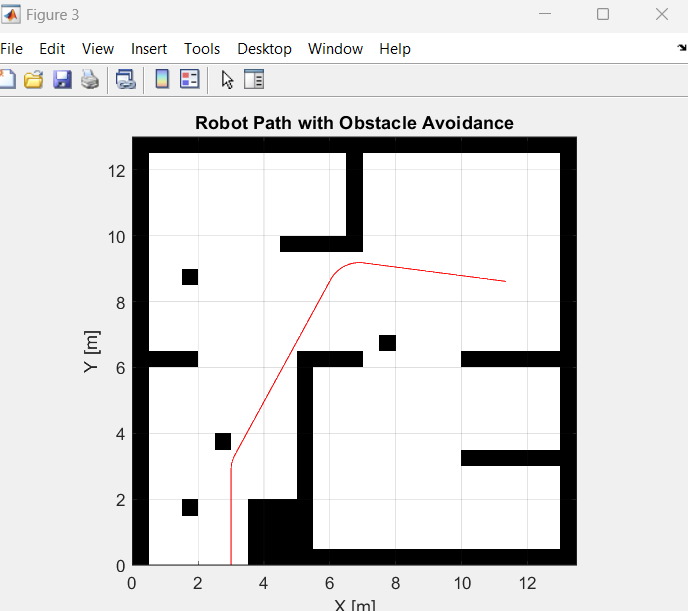
****

****

**** 

****

****

****

**Initialization and Setup**

**Clearing the Workspace:**

• c; clear;: All commands in the command window are cleared and all workspace variables cleared so that nothing is carried over from past computations.

**Open Example**

• openExample('robotics/PathPlanningExample');: A MATLAB example, which is set to be used as a starter guide, is opened to visualize and develop an intuitive feeling about basic ideas of path planning.

• load exampleMaps.mat;: Load a map with given obstacles to avoid doing preliminary work in setting up the problem.

**Robot Model:**

• robot = differentialDriveKinematics(.); : Defines a differential drive kinematic model, which will have physical parameters of the robot such as the wheel radius, track width, and maximum wheel speeds. This model is necessary for simulating its movement.

**•** initialPose = [3; 0; pi/2]; : Defines the position of the robot in the world frame (x, y) and the angle of orientation of the robot in that space, theta.

**Simulation Parameters:**

• end=30; dt = 0.1; t= 0:dt:tend;: A time horizon for the simulation, step size and generates a vector of time points for the time loop.

• pose(:, 1) = initialPose;: At each time step, pose matrix in intended to hold the pose of the robot during the run.

**Map and Obstacles:**

• map = binaryOccupancyMap(simpleMap, 2); : This is creating a binary occupancy map, where the environment can be considered as a grid of cells. Every cell represents free space or occupied space (0 or 1).

• setOccupancy(map, [2 2; 3 4; 5 2; 8 7; 2 9], 1);: This designates certain cells as occupied in the map, thus, mimicking the placement of obstacles in the given environment.

**Sensor and Control:**

**Sensor Model:**

• sensor = rangeSensor(.);: There is a creation of the range sensor model-a the real sensor, like lidar. The sensor range for the sensor measures distances to obstacles within a specified area of view.

• safeDistance = 0.5;: This is a safety distance threshold defined. If an obstacle is closer than this distance, the robot will move in evasive.

**Control Loop:**

• The for loop iterates over the time steps, simulating the robot's motion and sensor readings

Sensor Readings:

• [ranges, angles] = sensor(.);: It serves as a simulation of the measurements of the sensor by providing the current robot pose and map to the sensor model. The ranges and angles denote the distances and angles to the detected obstacles.

**• Obstacle Avoidance:**

• if min(ranges) < safeDistance: Now, if any of the detected obstacle is closer to it than the safety distance.

• omega = -0.5; : If an obstacle is too close, then it turns away to avoid it, by giving a negative angular velocity.

• omega = 0;: If no constraint exists within the safety distance, the robot maintains its heading.

**Robot Movement:**

• vel = derivative(robot, pose(:, i-1)', [v omega]); This line of code computes the robot linear and angular velocity using the differential drive kinematic model combined with the current control inputs-the linear velocity v and angular velocity omega.

• pose(:, i) = pose(:, i-1) + vel \* dt;: Updates the next time step of robot's pose using Euler's integration method.

**Monte Carlo Localization Configuration**

**Declaration of Variables:**

• odometryModel = odometryMotionModel;: Configured the odometry motion model.

• odometryModel.Noise = [0.2 0.2 0.2 0.2];: Declared the noise parameters of the motion model.

• rangeFinderModel = likelihoodFieldSensorModel;: Configured the likelihood field sensor model.

• rangeFinderModel.SensorLimits = [0.45 8];: Declared the sensor limits of the range finder model.

* rangeFinderModel.Map = map;: Assigns the map to the sensor model.

**Monte Carlo Localization Initialization:**

* numParticles = 1000;: Sets the number of particles for the localization algorithm.
* initialCovariance = eye(3);: Initializes the covariance matrix.
* mcl = monteCarloLocalization(.);: Creates the Monte Carlo Localization object with the specified initial pose, covariance, and particle limits.
* mcl.MotionModel = odometryModel;: Assigns the motion model to the MCL object.
* mcl.SensorModel = rangeFinderModel;: Assigns the sensor model to the MCL object.
* mcl.GlobalLocalization = false;: Turns off the global localization
* mcl.UseLidarScan = true;: Enables lidar scans

**Simulation Loop**

**• Sensor Readings:**

* [ranges, angles] = sensor(pose(:, i-1)' + [0.4 0.4 pi/2], map);: Reads sensor data based on the pose of the robot and the map.
* scan = lidarScan(ranges, angles);: Converts the data read from the sensors to a lidar scan object.

**• Obstacle Avoidance:**

* if min(ranges) < safeDistance: Determines if there is an obstacle that is close enough.
* omega = -0.5;: The robot's angular velocity is tweaked to avoid obstacles.

**• Updated Robot Pose:**

* new = derivative(robot, pose(:, i-1)', [v omega]);: Computes the velocity of the robot.
* pose(:, i) = pose(:, i-1) + vel \* dt;: Advances the robot pose

**• Monte Carlo Localization Update:**

* odometryPose = pose(:, i);: Gets the updated pose.
* [isUpdated, estimatedPose(:, i), covariance] = mcl(odometryPose, scan);: Updates the MCL with the new pose and sensor data.
* **Visualization Setup:**
  + figure;: Creates a new figure window for visualization.
  + TrackWidth = 0.5; framesize = 0.8 \* TrackWidth;: Sets parameters for visualizing the robot's shape and size.
  + vizRate = rateControl(1/dt);: Creates a rate control object to limit the visualization frame rate, ensuring smooth animation.
* **Plotting Robot and Map:**
  + show(map);: Displays the binary occupancy map.
  + T1 = [pose(1, i); pose(2, i); 0];: Defines the translation matrix for the robot's position.
  + plotRot = axang2quat([0 0 1 pose(3, i)]);: Defines the rotation matrix for the robot's orientation.
  + plotTransforms(...): Plots the robot's position and orientation using a 3D model.
  + light; xlim([0 13]); ylim([0 13]); waitfor(vizRate);: Sets lighting conditions, axis limits, and waits for the visualization rate to control the animation speed.
* **Plotting the Path:**
  + figure; show(map); hold on; plot(pose(1, :), pose(2, :), 'r'); hold off;: Plots the robot's trajectory on top of the map for analysis.

**Methodology and Algorithm**

* **Methodology:** This code utilizes MCL to localize the robot in a given map. It hybridizes probabilistic models of motion and sensor measurements for spatial localization in dynamically changing, noisy environments.
* **Algorithm:**

Sensor Reading and Obstacle Detection:

This robot employs a simulated range sensor that measures ranges to obstacles coming into view within its line of sight.

These measurements are then used to update the belief of the robot about its position, following the method of Monte Carlo Localization**.**

**Obstacle Avoidance Maneuver:**

**If there is an obstacle sensed to be within a safety distance:**

* The robot carries out a simple avoidance maneuver through the adjustment in its angular velocity as expressed by omega.

**If there are no obstacles sensed within the safety distance:**

* The robot continues its way heading in the same direction with some fixed linear velocity as expressed by v.

**Robot Motion Update:**

* The new position and orientation of the robot are calculated by a differential drive kinematic model.
* The model links the wheel velocities with the linear and angular velocities of the robot.
* The updates of the robot pose can be performed based on the integration method of Euler's formula of a small time step.

**Monte Carlo Localization:**

* The MCL algorithm used to update the estimate of the robot's pose on the basis of sensor readings and motion model.
* The particles are the representations of all probable poses of the robot, and the weights of these particles are updated according to the probabilities of the current sensor measurements.
* Resampling focuses on the most likely pose and discards unlikely poses by performing the algorithm.

**Visualization:**

* Current position, orientation as well as estimated pose of the robot together with the map and obstacles in graphic visualization.

**Important Points:**

* Monte Carlo Localization: Using particles to represent all the robot's possible poses; it provides the probabilistic update according to sensor and motion models.
* Reactive Obstacle Avoidance: The robot responds to any detected obstacle within its safety distance by adjusting the angular velocity.
* Differential Drive Kinematic Model: To compute the motion of the robot using the velocities of its wheels.
* Numerical Integration: Update the pose of the robot over time using Euler's method.
* Visualization: The motion of the robot, the estimated pose, and interaction with its environment can be visualized.

**Limitations and Future Improvements:**

* Simple Avoidance Maneuver: The moving strategy used within the existing avoidance algorithm is naive and could not be optimal in complex situations. Advanced algorithms include dynamic window approach or potential field methods.
* Sensor Model: Simulated sensor has limited range and accuracy. In reality, sensors would be modeled differently, which might impact the performance of this robot.
* Dynamic Environments: The technique is probably not suited for very dynamic environments where moving obstacles change positions at very high speeds. Techniques like MPC or RL should be more appropriate.
* Localization Accuracy. The accuracy of the MCL algorithm relies heavily on the number of particles and the quality of sensor and motion models. Adaptive techniques may be used to optimize the distribution of the particles and maximize localization accuracy.